HW1

Problem 1.

Solution

1. First, we can find intervals near $x = 0.95$ that bracket a root. By dividing the interval $[0.8, 0.98]$ into 1000 parts and testing each interval, we can obtain a list of intervals containing the roots.
2. For each interval identified in the previous step, apply the bisection method to find the approximate coordinates of the root.
3. Compare the distance between each root and $x = 0.95$, and choose the nearest four roots.

Code

```
import numpy as np
import math

def f(x):
    # the function given by problem 1
    return x * math.sin((x - 2) / (x - 1))

def find_intervals(f, a, b, steps):
    # divide interval [a,b] into #steps pieces, check whether each interval bracket the root
    x_values = np.linspace(a, b, steps)
    y_values = [f(x) for x in x_values]
    intervals = []
    for i in range(steps - 1):
        if y_values[i] * y_values[i + 1] < 0:
            intervals.append((round(x_values[i],5), round(x_values[i + 1],5)))
    return intervals

def bisection(f, a, b, tol, cnt):
    # approximates a root, R, of f bounded
    # by a and b to within tolerance
    # | f(m) | < tol with m the midpoint
    # between a and b Recursive implementation

    # check if a and b bound a root
    if np.sign(f(a)) == np.sign(f(b)):
        raise Exception(
         "The scalars a and b do not bound a root")

    # get midpoint
    m = (a + b)/2

    if np.abs(f(m)) < tol:
        # stopping condition, report m as root
        return m, cnt
    elif np.sign(f(a)) == np.sign(f(m)):
        # case where m is an improvement on a.
        # Make recursive call with a = m
        return bisection(f, m, b, tol, cnt+1)
    elif np.sign(f(b)) == np.sign(f(m)):
        # case where m is an improvement on b.
        # Make recursive call with b = m
        return bisection(f, a, m, tol, cnt+1)
```

```
# Find intervals around 0.95 where the function changes sign
intervals = find_intervals(f, 0.8, 0.98, 1000)
roots, iterations = [], []
for a, b in intervals:
    root, cnt = bisection(f, a, b, 10e-9, 0)
    roots.append(root)
    iterations.append(cnt)
distance = list(enumerate([abs(root - 0.95) for root in roots]))
distance_sorted = sorted(distance, key=lambda x:x[1])

count = 0
for i, j in distance_sorted:
    count += 1
    if(count > 4):
        break
    print(f"{count} nearest root to x = 0.95 is at x = {round(roots[i],5)}, distance to 0.95 
is {round(j,5)}, iteration times = {iterations[i]}")
```

Result

```
❯ python p1.py
1 nearest root to x = 0.95 is at x = 0.95236, distance to 0.95 is 0.00236, iteration times = 16
2 nearest root to x = 0.95 is at x = 0.94398, distance to 0.95 is 0.00602, iteration times = 17
3 nearest root to x = 0.95 is at x = 0.95856, distance to 0.95 is 0.00856, iteration times = 21
4 nearest root to x = 0.95 is at x = 0.96334, distance to 0.95 is 0.01334, iteration times = 21
```

Problem 2.

Code

```
import numpy as np
import math

def f(x):
    # the function given by problem 1
    return x * math.sin((x - 2) / (x - 1))

def find_intervals(f, a, b, steps):
    # divide interval [a,b] into #steps pieces, check whether each interval bracket the root
    x_values = np.linspace(a, b, steps)
    y_values = [f(x) for x in x_values]
    intervals = []
    for i in range(steps - 1):
        if y_values[i] * y_values[i + 1] < 0:
            intervals.append((round(x_values[i],5), round(x_values[i + 1],5)))
    return intervals

def secant(f, x0, x1, e, N):
    cnt = 1
    condition = True
    while condition:
        if f(x0) == f(x1):
            print('Divide by zero error!')
            break

        x2 = x0 - (x1 - x0) * f(x0) / (f(x1) - f(x0))
```

```
        x0 = x1
        x1 = x2
        cnt = cnt + 1

        if cnt > N:
            print('Not Convergent!')
            break

        condition = abs(f(x2)) > e
    return x2, cnt

# Find intervals around 0.95 where the function changes sign
intervals = find_intervals(f, 0.8, 0.98, 1000)
roots, iterations = [], []
for a, b in intervals:
    root, cnt = secant(f, a, b, 10e-9, 100)
    roots.append(root)
    iterations.append(cnt)
distance = list(enumerate([abs(root - 0.95) for root in roots]))
distance_sorted = sorted(distance, key=lambda x:x[1])

count = 0
for i, j in distance_sorted:
    count += 1
    if(count > 4):
        break
    print(f"{count} nearest root to x = 0.95 is at x = {round(roots[i],5)}, distance to 0.95 is {round(j,5)}, iteration times = {iterations[i]}")
```

Result

```
❯ python p2.py
1 nearest root to x = 0.95 is at x = 0.95236, distance to 0.95 is 0.00236, iteration times = 4
2 nearest root to x = 0.95 is at x = 0.94398, distance to 0.95 is 0.00602, iteration times = 4
3 nearest root to x = 0.95 is at x = 0.95856, distance to 0.95 is 0.00856, iteration times = 4
4 nearest root to x = 0.95 is at x = 0.96334, distance to 0.95 is 0.01334, iteration times = 4
```

Compared to the result in problem 1, the iteration time is about 12~17 fewer when we use secant method.

Problem 3.

a. When using the bisection method, we can easily find roots where the function changes signs. So roots at $x = 2$ can get with bisection but roots at $x = 4$ can't. Because roots at $x = 2$ have multiplicity 3, indicating that there is a sign change around it. However, we cannot be able to get the root at $x = 4$ because there's no sign change around it, the function remains positive as x approaches 2 from either side due to the odd multiplicity of the root.

b. The secant method does not require a sign change to converge to a root. Hence, it could be used to approximate both roots at $x = 2$ and $x = 4$.

c. All three method would get root at $x = 2$.

```
import numpy as np

def f(x):
    return pow((x - 2),3) * pow((x - 4), 2)

def bisection(f, a, b, tol, cnt):
    # approximates a root, R, of f bounded
    # by a and b to within tolerance
    # | f(m) | < tol with m the midpoint
```

```
    # between a and b Recursive implementation

    # check if a and b bound a root
    if np.sign(f(a)) == np.sign(f(b)):
        raise Exception(
         "The scalars a and b do not bound a root")

    # get midpoint
    m = (a + b)/2

    if np.abs(f(m)) < tol:
        # stopping condition, report m as root
        return m
    elif np.sign(f(a)) == np.sign(f(m)):
        # case where m is an improvement on a.
        # Make recursive call with a = m
        return bisection(f, m, b, tol, cnt+1)
    elif np.sign(f(b)) == np.sign(f(m)):
        # case where m is an improvement on b.
        # Make recursive call with b = m
        return bisection(f, a, m, tol, cnt+1)

def secant(f, x0, x1, e):
    cnt = 1
    condition = True
    while condition:
        if f(x0) == f(x1):
            print('Divide by zero error!')
            break
        x2 = x0 - (x1 - x0) * f(x0) / (f(x1) - f(x0))
        x0 = x1
        x1 = x2
        cnt = cnt + 1
        condition = abs(f(x2)) > e
    return x2

def falsePosition(f, x0, x1 ,e):
    cnt = 1
    condition = True
    while condition:
        x2 = x0 - (x1 - x0) * f(x0) / (f(x1) - f(x0))
        if f(x0) * f(x2) < 0:
            x1 = x2
        else:
            x0 = x2
        cnt = cnt + 1
        condition = abs(f(x2)) > e
    return x2

a, b = 1, 5

root_1 = bisection(f, a, b, 10e-9, 0)
root_2 = secant(f, a, b, 10e-9)
root_3 = falsePosition(f, a, b, 10e-9)

print(f"root get by bisection: {root_1}")
```

```
print(f"root get by secant method: {root_2}")
print(f"root get by false position: {root_3}")
```

```
❯ python p3_c.py
root get by bisection: 2.0
root get by secant method: 2.0
root get by false position: 2.0
```

Problem 4.

Code

```
import cmath
import numpy as np

def mullers_method(f, x0, x1, x2, tol=1e-5, max_iterations=100):
    h1 = x1 - x0
    h2 = x2 - x1
    delta1 = (f(x1) - f(x0)) / h1
    delta2 = (f(x2) - f(x1)) / h2
    d = (delta2 - delta1) / (h2 + h1)
    i = 0

    while i <= max_iterations:
        b = delta2 + h2 * d
        D = cmath.sqrt(b ** 2 - 4 * f(x2) * d)
        if abs(b - D) < abs(b + D):
            E = b + D
        else:
            E = b - D
        h = -2 * f(x2) / E
        x3 = x2 + h
        if abs(h) < tol:
            return x3
        x0, x1, x2 = x1, x2, x3
        h1 = x1 - x0
        h2 = x2 - x1
        delta1 = (f(x1) - f(x0)) / h1
        delta2 = (f(x2) - f(x1)) / h2
        d = (delta2 - delta1) / (h2 + h1)
        i += 1
    raise ValueError(f"The method did not converge after {max_iterations} iterations.")

def fa(x):
    return 4*x**3 - 3*x**2 + 2*x - 1

def fb(x):
    return x**2 + np.exp(x) - 5

x0, x1, x2 = 0, 0.5, 1
root_a = mullers_method(fa, x0, x1, x2).real

x0, x1, x2 = -3, -2, -1
root_b1 = mullers_method(fb, x0, x1, x2).real

x0, x1, x2 = 0, 1, 2
root_b2 = mullers_method(fb, x0, x1, x2).real
```

```
print(f"Root in problem a is at x = {round(root_a, 5)}")
print(f"Root in problem b is at x = {round(root_b1, 5)} and x = {round(root_b2, 5)}")
```

Result

```
❯ python p4.py
Root in problem a is at x = 0.60583
Root in problem b is at x = -2.21144 and x = 1.24114
```

Problem 5.

a. Shown below

b. Shown below

c. Rearrange $g(x) = \ln(2x^2) = \ln 2 + 2\ln x$

Code

```
import cmath
import numpy as np

def fixed_point_method(x, g, tol=1e-9):
    while(abs(g(x) - x) > tol):
        x = g(x)
    return x

def g_pos(x):
    return cmath.sqrt(np.exp(x) / 2)

def g_neg(x):
    return -cmath.sqrt(np.exp(x) / 2)

def g_c(x):
    return cmath.log(2) + 2 * np.log(x)

root_a1 = fixed_point_method(0, g_pos).real
root_a2 = fixed_point_method(0, g_neg).real

root_b1 = fixed_point_method(2.5, g_pos).real
root_b2 = fixed_point_method(2.5, g_neg).real
root_b3 = fixed_point_method(2.7, g_pos).real
root_b4 = fixed_point_method(2.7, g_neg).real

root_c = fixed_point_method(2.5, g_c).real

print(f"Problem A:\nRoot near 1.5: x = {round(root_a1, 5)},\n Root near -0.5: x = {round(root
_a2, 5)}")
print(f"\nProblem B:\n x0 = 2.5 postive root: x = {round(root_b1, 5)}, negative root: {round
(root_b2, 5)}")
print(f" x0 = 2.7 postive root: x = {round(root_b3, 5)}, negative root: {round(root_b4, 5)}")
print(f"\nProblem C:\nroot at x = {round(root_c, 5)}")
```

Result

```
Problem A:
 root near 1.5: x = 1.48796,
 root near -0.5: x = -0.53984

Problem B:
 x0 = 2.5 postive root: x = 1.48796, negative root: -0.53984
 x0 = 2.7 postive root: x = inf, negative root: -0.53984

Problem C:
 root at x = 2.61787
```

Problem 6.

Code

```
import numpy as np

def Newton_method(x, f, f_prime, tol = 10e-9):
    while(abs(f(x)) > tol):
        x = x - f(x) / f_prime(x)
    return x

f = lambda x: x**2 + np.cos(x)**4 - x - 2
f_prime = lambda x: 2 * x - 3 * np.cos(x)**3 * np.sin(x) - 1
root = Newton_method(0, f, f_prime)
print(round(root, 5))
```

Result

```
❯ python p6.py
root is at x = -0.96442
```